Chapter 3

Type inference

We began Chapter 2 with the observation that the need to annotate every variable with its type makes programming in System Fω rather inconvenient. In contrast it is often possible to write programs in OCaml without specifying any types at all. For example, here is the the result of entering the composition function at the OCaml top level:

```
# fun f g x -> f (g x);;
- : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Comparing the code that we entered with the type that OCaml prints, we see that OCaml has determined the following:

- `f` has type `'a -> 'b`
- `g` has type `'c -> 'a`
- `x` has type `'c`
- the result of the function has type 'b

Although we didn't specify the types of the variables `f`, `g` or `x`, OCaml deduced that `f` and `g` have function type, that the output of `g` has the same type (`'a`) as the input of `f`, and that `x` has the same type (`'c`) as the input to `g`.

The process of determining a suitable type from an unannotated expression is known as *type inference* or type reconstruction. As we shall see, the standard algorithms for type inference have a number of appealing properties which have led to their adoption in many typed functional programming languages. Perhaps most importantly, the types that they infer are guaranteed to be the best possible types (in a sense which we shall make more precise in Section 3.3.2), so we may be confident that we do not lose generality when we allow type inference to do its job. The widespread adoption of type inference may be attributed both to these properties and to the fact that it is able to support many constructs beyond the simple abstractions and applications in our example above, including a form of polymorphism, algebraic data types, recursion (with some restrictions) and imperative features such as mutable references.

3.1 Background

The appeal of type inference is easy to understand. Types give structure to programs and can prevent many errors from occurring, but type annotations can make programs quite difficult to read, as we saw in Chapter 2. If we could somehow combine the succinctness of annotation-free code with the safety and expressiveness of System Fω then we would have the best of both worlds.

Unfortunately, it turns out that the problem of reconstructing a type for an unannotated System Fω program — or even for an unannotated System F program — is undecidable. For type inference to be consistent and reliable we must introduce a number of restrictions on exactly where polymorphism can occur. In particular, we will only allow variables bound with **`let`**, not variables bound with **`fun`**, to have polymorphic types, and we must place corresponding restrictions on the places where quantifiers can occur. Whereas System F allow quantified types such as $\forall\alpha.\alpha \to \alpha$ to occur anywhere that types can occur, in order to support type inference we must restrict types to *prenex* form — that is, we will only allow quantifiers at the outermost level.

3.2 The calculus

We will present the type inference algorithm using the variant of $\lambda^{\to}$ shown in Figures 3.1–3.5. Since much of the calculus is now familiar we will remark only on the changes.

In addition to base types and function types we have type variables (Figure 3.2), as in System F. However, unlike System F our calculus does not support quantifiers in types. Instead, we introduce a new category of *schemes* (Figure 3.3). A scheme $\forall\bar{\alpha}.A$ is conceptually similar to a type, but supports quantification over multiple variables — we write $\bar{\alpha}$ to denote a collection of type variables $\alpha_1, \alpha_2, \ldots \alpha_n$. Further, the body of a scheme is not itself a scheme, but a type: we cannot build schemes from other schemes. In fact, schemes occur only in the environment: the environment rules (Figure 3.4) now support associating term variables with schemes, rather than with types.

Finally, there are four rules for constructing terms (Figure 3.5):

There is a new rule for `let` bindings, scheme-intro, which tells us that if `M` has type `A` in an environment Γ then we can build a scheme from `A` by quantifying over those free variables that do not occur in the environment and use the scheme to type a second term `N` in an extended environment. The type of `N` becomes the type of the whole term `let x = M in N`.

We have renamed the variable rule to scheme-elim to reflect the fact that it is at uses of variables that we turn schemes back into types. If a variable `x` is associated with the scheme $\forall\bar{\alpha}.A$ in an environment then we may give `x` the type that results from *instantiating* the bound variables $\bar{\alpha}$ with with types $\bar{B}$. The tvar rule from Chapter 2 is a special case of scheme-elim where $\bar{\alpha}$ is the empty collection.

The $\to$-intro rule is largely similar to the version we saw in Chapter 2, but

$$\frac{}{* \text{ is a kind}}\ *\text{-kind}$$

Figure 3.1: The kind rule

$$\frac{}{\Gamma \vdash \mathcal{B} :: *}\ \text{kind-}\mathcal{B} \qquad \frac{\Gamma \vdash A : * \quad \Gamma \vdash B :: *}{\Gamma \vdash A \to B :: *}\ \text{kind-}\to$$

$$\frac{\alpha \in \Gamma}{\Gamma \vdash \alpha :: *}\ \text{kind-}\alpha$$

Figure 3.2: Type formation rules

$$\frac{\Gamma, \bar{\alpha} \vdash A :: *}{\Gamma \vdash \forall \bar{\alpha}.A \text{ is a scheme}}\ \text{scheme}$$

Figure 3.3: Scheme rule

$$\frac{}{\cdot \text{ is an environment}}\ \Gamma\text{-}\cdot \qquad \frac{\Gamma \text{ is an environment} \quad \Gamma \vdash S \text{ is a scheme}}{\Gamma, x{:}S \text{ is an environment}}\ \Gamma\text{-}:$$

$$\frac{\Gamma \text{ is an environment}}{\Gamma, \alpha \text{ is an environment}}\ \Gamma\text{-}::$$

Figure 3.4: Environment rules

$$\frac{\Gamma \vdash M : A \qquad \bar{\alpha} \notin fv(\Gamma) \qquad \Gamma, x : \forall \bar{\alpha}.A \vdash N : B}{\Gamma \vdash \text{let x } = M \text{ in } N : B}\ \text{scheme-intro} \qquad \frac{x : \forall \bar{\alpha}.A \in \Gamma \quad \Gamma \vdash B :: *\ (B \in \bar{B})}{\Gamma \vdash x : A[\bar{\alpha} := \bar{B}]}\ \text{scheme-elim}$$

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\ \to\text{-intro} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}\ \to\text{-elim}$$

Figure 3.5: Typing rules

$$
\begin{aligned}
\text{unify} &: \text{ConstraintSet} \rightarrow \text{Substitution} \\
\text{unify}(\emptyset) &= [] \\
\text{unify}(\{A = A\} \cup C) &= \text{unify}(C) \\
\text{unify}(\{\alpha = A\} \cup C) &= \text{unify}([\alpha \mapsto A]C)\circ[\alpha \mapsto A] \text{ when } \alpha \notin ftv(A) \\
\text{unify}(\{A = \alpha\} \cup C) &= \text{unify}([\alpha \mapsto A]C)\circ[\alpha \mapsto A] \text{ when } \alpha \notin ftv(A) \\
\text{unify}(\{A \rightarrow B = A' \rightarrow B'\} \cup C) &= \text{unify}(\{A = A', B = B'\} \cup C) \\
\text{unify}(\{A = B\} \cup C) &= FAIL
\end{aligned}
$$

Figure 3.6: The unification algorithm

there is now no type annotation associated with the variable `x`. It is the job of the inference algorithm to determine what the missing type should be.

The $\rightarrow$-elim rule is unchanged.

It is important to note that in contrast to the typing rules of Chapter 2, the rules in Figure 3.5 are not "algorithmic": we need some mechanism not given in the rules to find appropriate instantiations for the type `A` in the $\rightarrow$-intro rule.

3.3 An algorithm for type inference

Before we introduce the type inference algorithm we need some auxiliary concepts, *substitutions* and *constraints*, and an auxiliary algorithm, *unification*.

Substitutions *Substitutions* play a central role in the algorithm. A substitution is a map from variables to types. We write

$\{a_1 \mapsto A_1,\ a_2 \mapsto A_2,\ \dots\ a_n \mapsto A_n\}$

to denote substitutions from variables a_1, a_2, ... a_n to types A_1, A_2, ... A_n. For example, the substitution

$\{a \mapsto \mathcal{B},\ b \mapsto (\mathcal{B} \rightarrow \mathcal{B})\}$

maps `a` to $\mathcal{B}$ and `b` to $\mathcal{B} \rightarrow \mathcal{B}$.

Substitutions can be applied to types and larger objects which contain types, such as environments. Applying a substitution to a type involves replacing the free type variables in the type with the corresponding values in the map. For example, applying the substitution $\{a \mapsto \mathcal{B},\ b \mapsto (\mathcal{B} \rightarrow \mathcal{B})\}$ to the type $a \rightarrow b \rightarrow a$ produces the type $\mathcal{B} \rightarrow (\mathcal{B} \rightarrow \mathcal{B}) \rightarrow \mathcal{B}$.

We write σA for the application of the substitution σ to the type `A`. If there is a substitution σ such that $\sigma A = B$ then we say that `B` is a *substitution instance* of `A`.

Constraints A constraint is a requirement that two types have a common substitution instance. Constraints have the form `A = B`. For example, the following are all constraints:

a = b
$a \to b$ = $\mathcal{B} \to b$
$\mathcal{B}$ = $\mathcal{B}$
$\mathcal{B}$ = $\mathcal{B} \to \mathcal{B}$

Some constraints are unsatisfiable. For example the last constraint above, $\mathcal{B}$ = $\mathcal{B} \to \mathcal{B}$ cannot be satisfied because there is no substitution that turns $\mathcal{B}$ into $\mathcal{B} \to \mathcal{B}$ or vice versa.

3.3.1 Unification

Unification is a partial function which either turns a set of constraints into a substitution or fails when it encounters unsatisfiable constraints.

The unification algorithm is shown in Figure 3.6. The function `unify` accepts a constraint set as input and produces a substitution (or fails). There are six cases:

1. If there are no constraints in the set the result is the empty substitution.

2. Unifying a type `A` with itself has no effect: the constraint is dropped.

3,4 Unifying a variable `a` with a type `A` builds a substitution which maps `a` to `A`. The notation $\sigma \circ \sigma'$ composes substitutions: applying the resulting substitution has the same effect as applying σ' then applying σ.

The check that `a` does not occur in the free type variables of `A` prevents the construction of recursive types.

5 Unifying types `A`→`B` and `A'`→`B'` proceeds by unifying `A` with `A'` and `B` with `B'`.

6 Attempting to unify any other types `A` and `B` causes unification to fail.

3.3.2 Algorithm J

The type inference algorithm described in this chapter was introduced by Robin Milner in a 1978 paper, *A theory of type polymorphism in programming*. Milner's paper presented two algorithms for reconstructing types for unannotated expressions. Algorithm W is a purely functional algorithm which accumulates substitutions and carefully applies them at appropriate points. Algorithm J is an imperative algorithm which simulates W and uses a single global substitution which is updated as type inference progresses. Algorithm W is amenable to formal proofs, while Algorithm J is more efficient, and easier to understand. In this chapter we will focus on Algorithm J.

One important property of both Algorithm W and Algorithm J is that they infer *principal types*. A type `A` is principal for a term `M` if every other derivable type for `M` is a substitution instance of `A`. The principal types property assures us both that we can rely on the type inference algorithm to give us the best possible type, and that any valid annotation that we add to our terms will be compatible with the inferred type.

Figure 3.7 presents Algorithm J. The algorithm is defined by cases on the four syntax constructors. The binding forms `λx.M` and `let x = M in N` both extend the environment, but only `let` bindings introduce schemes with quantifiers. Schemes are instantiated at variable uses and unification takes place at function applications, where we must ensure that the argument supplied to a function is compatible with its inferred input type.

A worked example Let's use algorithm J to find a type for the expression

```
let apply = λf.λx.f x in let id = λy.y in apply id
```

in an empty environment. The algorithm begins at the fourth case, since we're typing a `let` expression. We must find

```
J(·, let apply = λf.λx.f x in let id = λy.y in apply id)
```

The first step is to type the right hand side `M`, i.e. `λf.λx.f x`:

```
J(·, λf.λx.f x)
```

The third case for J tells us how to type lambda abstractions. We pick a fresh variable $\mathtt{b}_1$ for the type of `f`, add a binding to the environment and type the body:

$\mathtt{J(\cdot,f:b_1,\ \lambda x.f\ x)}$

The body is also a lambda abstraction, so we do the same again:

$\mathtt{J(\cdot,f:b_1,x:b_2,\ f\ x)}$

Now we must type the application `f x`. We must first type the function term `f`. We use the case for variables, which is trivial in this instance since the scheme for `f` in the environment does not have any bound type variables to instantiate.

$\mathtt{J(\cdot,f:b_1,x:b_2,\ f)\ =\ b_1}$

The typing for the argument term `x` is similar:

$\mathtt{J(\cdot,f:b_1,x:b_2,\ x)\ =\ b_2}$

Continuing with the case for application, we must unify, using a fresh variable $\mathtt{b}_3$ for the result type:

$\mathtt{unify(\{b_1\ =\ b_2 \to b_3\})}$

The result of unification is the map

$\mathtt{\{b_1\ \mapsto\ b_2 \to\ b_3\}}$

and the type of `λx.f x` is $\mathrm{b}_2 \to \mathrm{b}_3$, and the type of `λf.λx.f x` is $\mathrm{b}_1 \to \mathrm{b}_2 \to \mathrm{b}_3$. Applying the substitution transforms this last type into $\mathtt{(b_2\ \to\ b_3)\ \to\ b_2\ \to\ b_3}$. Returning to the `let` case, we can now extend the environment and type the body. We first compute the free variables in the type and in the environment:

Algorithm J infers a type for an expression `M` in an environment Γ, if there is such a type. When the algorithm succeeds in producing a type `A`, the type is guaranteed to be *principal*: that is, every other valid type for `M` in Γ is a substitution instance of `A`.

```
J : Environment × Expression → Type
```

The algorithm operates on a single global substitution `E`.
There are four cases, corresponding to the four term constructors.

```
J (Γ, x) = A[$\bar{\alpha}$:=$\bar{b}$]
         where Γ(x) = $\forall\bar{\alpha}.A$
         and $\bar{b}$ are fresh
```

The first case deals with variables. A variable `x` is given a type by *instantiating* the scheme $\forall\bar{\alpha}.A$ associated with `x` in Γ — that is, by replacing the quantified variables $\bar{\alpha}$ with fresh variables $\bar{b}$, which may later be unified with other types.

```
J (Γ, M N) = $b$
         where A = J (Γ, M)
         and B = J (Γ, N)
         and unify' ({A = B → b}) succeeds
         and b is fresh
```

The second case deals with applications. An application `M N` is given a type by finding types `A` and `B` for `M` and `N`, then unifying `A` with a function type whose argument type is `B`. The `unify'` procedure (not shown) calls `unify` to build a substitution from the constraint set argument, then applies the substitution to `E` to produce the new value of `E`.

```
J (Γ, λx.M) = b → A
          where A = J (Γ, x:$b$, M)
          and b is fresh
```

The third case deals with function bindings. A function `λx.e` is assigned a type by choosing a fresh variable b for the argument, adding the argument to the environment, then inferring a type `A` for the body `M`. The fresh variable may be unified with a concrete type if `x` is used in `M`.

```
J (Γ, let x = M in N) = B
          where A = J (Γ, M)
          and B = J (Γ, x:$\forall\bar{\alpha}.A$, N)
          and $\bar{\alpha}$ = ftv(A) \ ftv(Γ)
```

The fourth case deals with `let` bindings. A let binding `let x = M in N` is given a type by first finding a type `A` for `M`, then generalising `A` by quantifying over any free type variables $\bar{\alpha}$ in `A` which do not appear in Γ. The type `B` of the body `M` is then determined in an environment extended with `x:`$\forall\bar{\alpha}.A$.

Figure 3.7: Algorithm J

```
ftv((b₂ → b₃)→ b₂ → b₃) = {b₂, b₃}
ftv(·) = {}
```

So `ftv(A)\ ftv(Γ)` is `{b₂, b₃}` and we extend the environment with a scheme that quantifies over b_2 and b_3. We must next find

```
J(·,apply:∀α₂α₃.(α₂ → α₃)→ α₂ → α₃, let id = λy.y in apply id)
```

Once again, we first type the right hand side `λy.y`

```
J(·,apply:∀α₂α₃.(α₂ → α₃)→ α₂ → α₃, λy.y)
```

We pick a fresh type variable b_4 for `y` and type the body:

```
J(·,apply:∀α₂α₃.(α₂ → α₃)→ α₂ → α₃,y:b₄, y) = b₄
```

So the type of `λy.y` is $b_4 \to b_4$. We must generalize the type and then type the body of the `let`. We have

```
ftv(b₄ → b₄) = {b₄}
ftv(·,apply:∀α₂α₃.(α₂ → α₃) → α₂ → α₃) = {}
```

so we quantify over b_4 and add the scheme to the environment. We then have

```
J(·, apply:∀α₂α₃.(α₂ → α₃) → α₂ → α₃,id:∀α₄.α₄ → α₄, apply id)
```

Typing the left and right hand sides of the application gives

```
J(·, apply:∀α₂α₃.(α₂ → α₃) → α₂ → α₃,id:∀α₄.α₄ → α₄, apply)
  = (b₅ → b₆)→ b₅ → b₆
J(·, apply:∀α₂α₃.(α₂ → α₃) → α₂ → α₃,id:∀α₄.α₄ → α₄, id)
  = b₇ → b₇
```

We choose a fresh variable b_8 for the result of the application and unify to build a substitution:

```
  unify ({(b₅ → b₆)→ b₅ → b₆ = (b₇ → b₇) →b₈})
= unify ({b₅ → b₆ = b₇ → b₇,
          b₅ → b₆ = b₈})
= unify ({b₅ = b₇,
          b₆ = b₇,
          b₅ → b₆ = b₈})
= {b₅ ↦ b₇, b₆ ↦ b₇, b₈ ↦ b₅ → b₆}
```

Applying the substitution to b_8 gives $b_7 \to b_7$, which is the inferred type for the whole expression.

3.4 Type inference in practice

Up to this point we have focused on the core type inference algorithm. There are a number of additional considerations when programming in a language with type inference.

3.4.1 Turning `let` bindings into function arguments

System Fω has the appealing property that it is always possible to generalize a function by turning its free identifiers into parameters. Unfortunately, OCaml does not share this property: whether an identifier used inside a function can be turned into a parameter depends on whether the identifier is used polymorphically. For example, consider the function that “doubles” each of a pair of lists using the list append operator `@`:

```
let double2 ((l : int list), (r : string list))
  = (l @ l, r @ r)
```

The append operator has a type which allows it to be used both on lists of integers and lists of strings:

```
val (@) : 'a list -> 'a list -> 'a list
```

We might like to generalize the function by turning the append operation into a parameter as follows:

```
let double2 (@) ((l : int list), (r : string list))
  = (l @ l, r @ r)
```

This definition is rejected by OCaml because it would require the first parameter of `double` to be given a polymorphic type, which is incompatible with type inference. On the other hand, the following definition is accepted, since the parameter `@` is only used at a single type in the body of the function:

```
let double2 (@) ((l : int list), (r : int list))
  = (l @ l, r @ r)
```

As this example illustrates, the restriction to prenex polymorphism brings a kind of non-uniformity to languages with type inference, since only certain types of values can be lambda-abstracted.

Chapter 6 describes some of the features available in OCaml that make it possible to work around these limitations.

3.4.2 Type inference and recursion

Extending the calculus in Figures 3.1–3.5 to handle recursion is straightforward. The following rule supports an OCaml-style `let rec` binding:

$$\frac{\Gamma, x{:}A \vdash M : A \qquad \bar{\alpha} \notin fv(\Gamma) \qquad \Gamma, x : \forall\bar{\alpha}.A \vdash N : B}{\Gamma \vdash \text{let rec x = M in N} : B}\ \text{let-rec}$$

Unlike the rule for `let` binding, the environment is extended both when typing `N` and when typing `M`, since `let rec` brings `x` into scope in both expressions.

However, the terms `M` and `N` are not typed in the same environment. It is only when typing `N` that the free variables in the type `A` are generalized. When we begin typing `M` we do not yet know the type `A`, and so it is not possible to determine its free variables. In concrete terms this means that the type inference algorithm will reject functions that are used polymorphically in their own definitions.

Let's look at an example. In Chapter 2 we saw the following data type definition:

```
type 'a perfect =
   ZeroP : 'a -> 'a perfect
 | SuccP : ('a * 'a) perfect -> 'a perfect
```

Even fairly simple functions over values of type `perfect` require polymorphic recursion. For example, consider the function that determines the depth of a tree:

```
let rec depthP = function
   ZeroP _ -> 0
 | SuccP p -> succ (depthP p)
```

We expect the function to have type `'a perfect -> int`, but the recursive call is given a value of type `('a * 'a) perfect`. Consequently, type inference fails and OCaml rejects the definition:

```
   Characters 67-68:
     | SuccP p -> succ (depthP p)
                               ^
   Error: This expression has type ('a * 'a) perfect
          but an expression was expected of type 'a perfect
          The type variable 'a occurs inside 'a * 'a
```

We can fix the problem by telling OCaml the expected type in advance — the equivalent of supplying a pre-generalized `A` to use when typing `M` in the let-rec rule:

```
let rec depthP : 'a. 'a perfect -> int = function
   ZeroP _ -> 0
 | SuccP p -> succ (depthP p)
```

While nested data types like `perfect` are fairly rare, polymorphic recursion is often needed when using GADTs (Chapter 8), which are increasingly common in OCaml programs.

3.4.3 Supporting imperative programming: the value restriction

A number of other OCaml features besides recursion have non-trivial interactions with type inference. In particular, the presence of mutable state requires changing the type inference algorithm to avoid unsoundness.

Mutable references in OCaml OCaml supports mutability via the **`mutable`** keyword, which can be attached to record fields. For example, here is the definition of the standard library `ref` type, which represents a mutable reference cell:

```
  type 'a ref = { mutable contents : 'a }
```

The standard library also provides a number of functions for manipulating reference cells. The `ref` function creates a fresh reference cell:

```
val ref : 'a -> 'a ref
```

The prefix operator `!` retrieves the current contents of a reference cell:

```
val ( ! ) : 'a ref -> 'a
```

The infix operator `:=` updates the contents of a reference cell:

```
val (:=) : 'a ref -> 'a -> unit
```

We can use this interface to construct an example which shows the unsoundness that results when the inference algorithm is applied to a language with mutable state:

```
let r = ref None in
   r := Some "boom";
   match !r with
     None -> ()
   | Some f -> f ()
```

The unmodified Algorithm J infers the scheme `'a.'a option ref` for the expression `ref None`. This scheme allows `r` to be instantiated at the type `string option ref` on the second line, which stores the value `Some "boom"` in the cell, and instantiated at the type `(unit -> unit) option ref` on the last line, which retrieves the contents of the cell and invokes it. Treating the string stored in the cell as a function causes the program to crash.

How can we characterise this unsoundness? If we view the inference algorithm in the light of what we know about System F then things become clear. In System F generalisation and instantiation appear as a type of function abstraction ($\Lambda\alpha$`.M`) and application (`M [`α`]`). If we view the inference algorithm as a program transformation which transforms terms to insert these introduction and elimination forms for polymorphism then the problem is apparent: changing `ref None` into $\Lambda\alpha$`.ref None` changes the order of evaluation, since $\Lambda\alpha$`.M` delays evaluation of the term `M`. This intuition suggests the solution: we should only allow generalization of a type when the associated term is already in value form — that is, when evaluating it has no effect.

This solution, was adopted by SML and by OCaml following a study in 1995 of several ML codebases by Andrew Wright. Wright discovered that almost all terms that were used polymorphically were already syntactic values, making the "value restriction" relatively benign. For those terms that are not in value form η-expansion often fixes the problem: we can turn

```
let f = g x in e
```

into

```
let f = fun y -> g x y in e
```

to recover the lost polymorphism. Of course, if `g x` does not have function type, or if the program depends on the existing evaluation order then this solution is not suitable, but Wright found almost no bindings that could not be straightforwardly fixed using this approach and the value restriction was soon adopted for both Standard ML and OCaml.

3.4.4 Relaxing the value restriction

Although Wright's study found few cases where the value restriction caused existing code to be rejected, there are situations where restricting generalisation to syntactic values is a source of inconvenience. For example, we might represent a bounded list type, which can hold no more than some specified number of elements, as a pair of the bound and the list:

```
type 'a blist = int * 'a list
```

It is straightforward to create empty `blist` values:

```
let bl5 : 'a blist = (5, [])
let bl3 : 'a blist = (3, [])
```

We might like to abstract empty `blist` creation with a function:

```
let empty_blist : int -> 'a blist = fun n -> (n, [])
```

However, the value restriction means that we lose polymorphism by abstracting. The following definitiion results in a non-generalized type for `bl5`, since the right hand side is not a syntactic value.

```
let bl5 = empty_blist 5
```

Since the expression does not have a function type it is not possible to recover the lost polymorphism by eta expansion.

As this example ilustrates, the simple check for whether an expression is a syntactic value is sometimes too crude a test. For this reason OCaml uses a refinement of the value restriction which assigns polymorphic types to more expressions. In order to understand OCaml's approach it is first necessary to understand the concepts of *covariance* and *contravariance.*

Covariant and contravariant type variables In a function type `s -> t` the types `s` and `t` play quite different roles: `s` indicates the type of values that are consumed by the function, whereas `t` indicates the type of values produced.

These different roles must form part of our reasoning when we consider the effects the changes in a function's behaviour have on callers of that function. We can always extend a function to accept a wider range of inputs without a negative impact on callers; similarly, it is always safe to restrict a function to produce a narrower range of outputs. However, in neither case is the converse true: either restricting the input set or expanding the output set breaks the "contract" between the existing function and its callers.

For simple function types like `s -> t` it is straightforward to identify inputs and outputs. However, we sometimes want to reason about more complex types, where it is helpful to generalize the idea of input and output to *contravariant* and *covariant* positions in a type. For example, consider the type

```
type 'a printer = 'a -> string
```

Since the type parameter `'a` is used as an input in the definition of `printer`, OCaml marks it as contravariant[1]. Contravriance acts like negation, so that a

[1]The contravariance flag is an internal feature that is not usually displayed

contravariant parameter of a type constructor that is itself in a contravariant position is covariant. For example, in the type

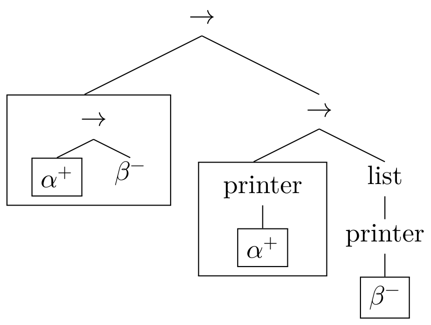
```
('a printer) printer
```

the type `'a` is marked as covariant.

Here is a larger example. The following type has two type variables `'a` and `'b` which respectively appear only in covariant and contravariant positions:

```
('a -> 'b) -> 'a printer -> 'b printer list
```

We can visualize the type as a tree, where each box represents a contravariant position:



Each type variable that is surrounded by an odd number of boxes is in a contravariant position, and every other type variable is covariant.

What does variance have to do with the value restriction? Since covariant positions represent outputs we know that if a type variable is only used in covariant positions it represents something that is only read, not written[2]. It is safe to generalize such variables because the unsound situation in Section 3.4.3, where a reference cell was modified at one type and accessed at another, can never occur.

Let's look at the `blist` example again in the light of this "relaxed value restriction". The expression

```
empty_blist 5
```

has type `'a blist`, which is equivalent to `int * 'a list`.In this type the variable `'a` occurs only covariantly, so it is safe to generalize, which is what OCaml does. On the other hand the parameter of the `ref` type constructor is considered *invariant* by OCaml, since it can be used for both input and output, so the expression

```
ref None
```

which has type `'a option ref` is not generalized.

3.4.5 Closing remarks

Historically type inference has played a large part in the development of OCaml and other ML-family languages, and there has been reluctance to add new

[2]The actual justification for generalizing covariant type variables is based on a technical argument about subtyping.

features for which types could not be inferred. In recent years type inference has played a less central role, as OCaml has acquired features like GADTs (Chapter 8) which do not support inference. This shift in focus has brought with it a number of smaller changes: for example, recent versions of OCaml use types to disambiguate different data constructors which have the same name.

In the next chapter we will consider the correspondence between propositions and types, and between programs and proofs. Viewed from this perspective, type inference seems a little strange. It is easy to see why we might want a proof system where we can specify propositions and have the computer synthesize proofs, but it is less clear why we might want to specify proofs and have the computer infer propositions. The more expressive types available in modern functional programming languages have less to do with simple data layout and more to do with structuring programs and describing behaviour. As types acquire this larger and more interesting role, we might start to question whether leaving them to computers to determine is the right decision after all.

3.5 Exercises

1. [★★] Step through the inference algorithm for the following expression

```
let k = λx.λy.x in
let i = λx.x in
  k k i
```

2. [★★★] What happens in the following example if the free variables from the environment are not removed when inferring a type for the `let` binding?

```
let f = λx.
          let f = λy.x in
          (f ⟨⟩) ⟨⟩
 in f ⟨⟩
```

   (Assume that ⟨⟩ is assigned the type `unit` and behaves like OCaml's unit value.)

3. [★★★] Give an example of an OCaml function that is rejected due to the lack of polymorphic types for function arguments.

4. [★★★] OCaml rejects the following function with an error:

```
let rec f = fun (x, y) -> f (y, (x, x))
```

   Add a type annotation (between `f` and `=`) that allows the function to pass OCaml's type checking.

5. [★★★] Give an example of a function that is excluded by OCaml's relaxed value restriction but that would behave correctly when executed.

6. [★★] The check in the unification algorithm that prevents the formation of recursive types can be disabled in OCaml by the command-line flag

`-rectypes`. Find a function that is accepted by OCaml when `-rectypes` is specified but rejected otherwise.

Further reading

- Milner's 1978 paper introduced type inference for polymorphic programs, and describes the algorithms W and J that form the basis of many type inference implementations:

  A theory of type polymorphism in programming
  Robin Milner
  Journal of computer and system sciences 17, 348–375 (1978)

- Milner's paper showed that Algorithm W is sound. The follow-up paper shows that Algorithm W is also complete — i.e. that it infers principal types.

  Principal type-schemes for functional programs
  Luis Damas and Robin Milner
  Principles of programming languages (1982)

- Wright's paper introducing the value restriction contains an interesting account of the history of combining type inference with imperative programming.

  Simple imperative polymorphism
  Andrew K. Wright
  Lisp and symbolic computation 8(4), 343-355 (1995)

- The relaxed value restriction used by OCaml is described in the following paper:

  Relaxing the value restriction
  Jacques Garrigue
  International symposium on functional and logic programming (2004)